An Efficient Asymmetric Distributed Lock for Embedded Multiprocessor Systems

Jochem H. Rutgers, Marco J.G. Bekooij, Gerard J.M. Smit
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
{j.h.rutgers, m.j.g.bekooij, g.j.m.smit}@utwente.nl

***Abstract*—Efficient synchronization is a key concern in an embedded many-core system-on-chip (SoC). The use of atomic read-modify-write instructions combined with cache coherency as synchronization primitive is not always an option for shared-memory SoCs due to the lack of suitable IP. Furthermore, there are doubts about the scalability of hardware cache coherency protocols. Existing distributed locks for NUMA multiprocessor systems do not rely on cache coherency and are more scalable, but exchange many messages per lock.**

This paper introduces an asymmetric distributed lock algorithm for shared-memory embedded multiprocessor systems without hardware cache coherency. Messages are exchanged via a low-cost inter-processor communication ring in combination with a small local memory per processor. Typically, a mutex is used over and over again by the same process, which is exploited by our algorithm. As a result, the number of messages exchanged per lock is significantly reduced. Experiments with our 32-core system show that when having locks in SDRAM, 35% of the memory traffic is lock related. In comparison, our solution eliminates all of this traffic and reduces the execution time by up to 89%.

I. Introduction

Although the world is moving towards hardware with tens of cores on a single chip, programming such device is still challenging. Prominent issues are parallelization of the application and synchronization between these parts.

In modern general purpose chips, synchronization is usually polling-based using atomic read-modify-write (RMW) operations as building blocks [1, 2], which are either hidden in a lock library or used by the programmer directly. RMW operations have a relatively low latency and are wait-free [3]. However, they require a cache coherent system, which is hard to realize in general and absent in Intel's 48-core SCC [4], for example. Without cache coherency, RMW operations induce traffic to external SDRAM, which has a limited bandwidth and high latency.

Hardware cache coherency or RMW instructions are not always applied in embedded systems, because of high hardware costs and the lack of IP [5]. Additionally, hardware cache coherency is unsupported for FPGA targets by common system-on-chip (SoC) design tools, such as Xilinx XPS and Altera SOPC Builder, as neither the MicroBlaze nor the Nios II support it. The alternative is to use a generic software implementation of a synchronization algorithm, like the bakery algorithm for mutexes [6]. Again, the SDRAM is then used for synchronization, which is a scarce resource, as a single bank is shared among many cores. The memory can be bypassed for synchronization completely when using a distributed lock via message-passing [7], which requires local memories and an inter-processor network.

In this paper, we evaluate a 32-core SoC architecture and show that adding a low-cost inter-processor communication ring for synchronization reduces the required SDRAM memory bandwidth. Additionally, an efficient distributed lock algorithm is presented that reduces the average latency of locking, by exploiting the locality of mutexes. As a result, the throughput and execution time of applications improve.

For our experiments, we built a 32-core shared-memory MicroBlaze system on a Virtex-6 FPGA, which is described in Section III. Using this platform, memory traffic of several standard applications is studied in Section IV, which shows that mutex operations contribute significantly to the memory traffic. In Section V, an additional interconnect and distributed lock implementation is added to the system, which relieves the memory and improves the performance, as discussed in Section VI. Section VII concludes the paper.

II. Related work

Besides using RMW operations, hardware support for synchronization can also be realized differently. Stoif et al. use a central memory controller where processors compete for protected memory regions [8]. The memory controller can also be extended to manage the state of synchronization data units that reside in the memory [9, 10]. Tumeo et al. implement synchronization using engines like the Xilinx mutex component [11]. Like fixed synchronization networks [12], all these hardware components have in common that the number of concurrent synchronization primitives is limited, where our solution scales naturally in software. Additionally, centralized units fundamentally introduce a bottleneck when the system scales to more cores.

Symmetric distributed algorithms require many messages to operate [7], because all nodes need to be informed separately. These algorithms are aimed for fault-tolerance, but as we do not assume a faulty device and message exchange is relatively expensive, the overhead is needlessly high. Yu and Petrov introduce a distributed lock component for every core, which all snoop synchronization messages from a bus [13]. Having such global bus, limits scalability—the paper presents experiments with only four cores.

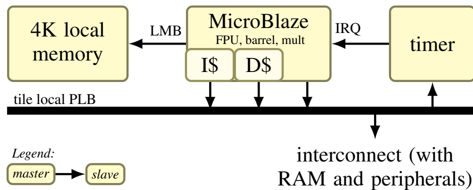
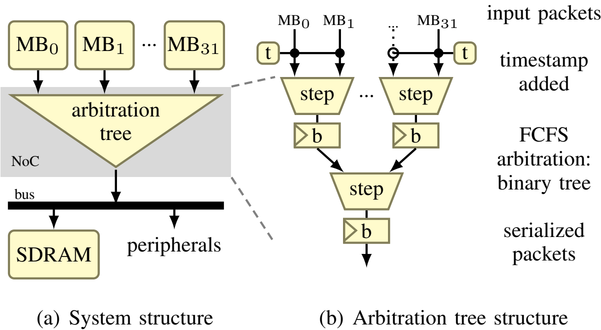


Fig. 1: MicroBlaze tile



(a) System structure (b) Arbitration tree structure

Fig. 2: Structure of the system

An asymmetric distributed mutual exclusion algorithm without experimental results has been proposed by Wu and Shu [14], using a central coordinator that forwards lock requests on mutexes, where processes form a distributed waiting queue. In this algorithm, queuing is done in a distributed manner, but locks are always sent back to the server on unlock. As we found out (see Section IV), mutexes are often reused by the same process. This still requires many messages, where our solution optimizes for this mutex locality and we add a quantitative evaluation.

Using local memories in a NUMA architecture for message-passing is a common approach [4, 15], but generic all-to-all communication is potentially expensive in hardware. However, Section V shows that our low-bandwidth, write-only ring implementation can be kept low-cost.

III. Platform: 32-core SoC

To investigate mutex behavior well, a realistic setup is required; accurate simulation is too slow and abstractions of a simulation model could hide incorrect assumptions on essential resource costs, for example. Therefore, we designed and built a 32-core many-core SoC to make realistic observations and run solid experiments. The SoC is based on MicroBlaze tiles, as depicted in Figure 1.

A. Tiles and Memory Hierarchy

Such tile contains the MicroBlaze core, a local memory that is used to boot from and store a few local variables of the kernel, and a PLB bus. To this bus, a timer is connected, which is the only interrupt source of the MicroBlaze.

The MicroBlaze has a 16 KB instruction cache and 8 KB data cache—both direct-mapped, using 8-word cache line size—connected to the local PLB bus. All code resides in main memory. The stack and heap of the core also reside in main memory and are cached. Since the MicroBlaze has no hardware cache coherency, inter-processor communication is done via a part of the main memory that is uncached. Shared data structures are also put in the uncached memory region.

The 256 MB main memory is accessible via an interconnect (see Figure 2(a)), which is quite expensive in terms of latency. The latency of a memory read in an idle system is about 30 clock cycles: 15 cycles to traverse the interconnect and 15 cycles for the memory controller to process the read.

B. Tree-shaped Interconnect with FCFS

Figure 2(b) shows the structure of the interconnect, which is a packet switched network-on-chip (NoC), with its main purpose to gain access to the main memory. The interconnect arbitrates requests of all 32 cores to a bus, where a memory controller and peripherals are connected to. The network supports read and write requests, which are issued by the processors. Every single request is separately packetized, containing one command, one address, and multiple data flits. Then, the packet will receive a timestamp for first-come-first-served (FCFS)-arbitration and a processor ID upon injection, which is sent along with the packet. The timestamp is generated locally to every core, by timers that are synchronized on global reset—but a few cycles deviation is acceptable, as long as local clocks are not drifting.

Next, the packets are sent through a binary arbitration tree that multiplexes n processors to one bus master, where every *step* in the tree does local arbitration of two inputs. A step lets the packet with the oldest timestamp precede. Rearbitration only happens between packets. After every step, a small buffer can be placed for shorter wires or left out for lower latency. Therefore, multiple packets can be 'in flight' towards the bus. By means of back-pressure, requests can be stalled by subsequent steps and the bus slave.

Finally, the response will be sent back via a similar tree, which routes the packet based on the processor ID, without the need of arbitration.

C. Synthesis for FPGA

The design has been implemented on a Xilinx Virtex-6 LX240-T FPGA, using the Xilinx ML605 development board. In total, the design uses 124,644 look-up tables (LUTs), 97,490 flip-flops (FFs) and 316 Block RAMs (BRAMs), where one MicroBlaze tile consists of 3,244 LUTs, 2,563 FFs, and 9 BRAMs, and the interconnect 9,511 LUTs and 5,759 FFs—roughly 7% of the total design. The remaining resources are used by the memory controller, DVI, Ethernet, UART and USB controller. All cores and the interconnect are clocked at 100 MHz.

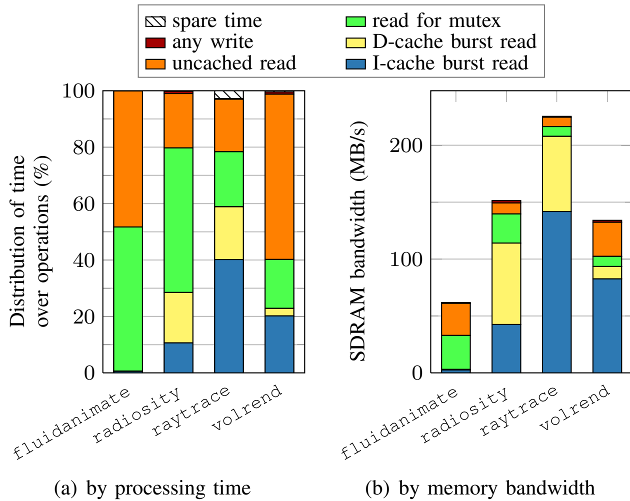
D. Software: Kernel and Applications

On every MicroBlaze in the SoC described above, a small stand-alone custom POSIX-like micro-kernel is running, which supports the `newlib` C library and implements the Pthread standard. The kernel can run multiple processes in

TABLE I: Applications

benchmark set	application	code size	mutexes
PARSEC[a]	`fluidanimate`	494 KB	4,403
SPLASH-2	`radiosity`	261 KB	26,034
	`raytrace`	244 KB	35
	`volrend`	421 KB	37

[a] All other applications of the set are not runnable, because of dependency problems or memory requirements.



(a) by processing time (b) by memory bandwidth

Fig. 3: Measured traffic on SDRAM controller

a multi-threaded fashion, without process migration between kernels. From a kernel point of view, daemons, servers, and (worker) threads from applications are just processes. MicroBlazes can only communicate via main memory (as the Pthread standard prescribes), and no RMW instructions are available. Hence, the `pthread_mutex_t` has been implemented using Lamport's bakery algorithm.

Using this platform, applications from the SPLASH-2 [16] and PARSEC [17] benchmarks have been run. Table I lists all applications used for the experiments.

IV. Memory and Mutex Profile

The memory controller has hardware support to measure memory traffic. Using this information and profiling data measured by the MicroBlaze and operating system, traffic streams of the applications can be identified, which is plotted in Figure 3. The figure distinguishes:

- 8-word burst *instruction cache reads* (bottom of chart);
- 8-word burst *data cache reads*;
- uncached word read, participating in locking a *mutex*;
- other *uncached* word read of shared memory;
- all 8-word burst and single word *writes*;
- all *spare* time the memory controller is idle (top).

Figure 3(a) depicts on which operations the time is spent by the memory controller. Since the spare time is (almost) zero, the controller is completely saturated and imposes a bottleneck in the system. The bandwidth usage corresponding

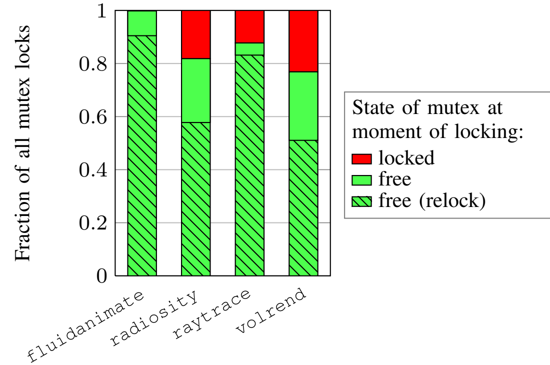


Fig. 4: Mutex locking behavior per application

to Figure 3(a) is shown in Figure 3(b). Although the controller is saturated for all applications, the bandwidth greatly differs, because word reads/writes take almost the same amount of time as burst reads/writes, but leave most of the potential bandwidth unused. SDRAM commands like precharge and refresh do not show up in the bandwidth, but do contribute to the latency of commands.

In both Figures 3(a) and 3(b), the writes are hardly visible as they occur relatively sporadically. Mutex operations contribute by 35% on average to the memory controller load and 18% to the bandwidth usage, surprisingly. Although the bakery algorithm relies on reads *and* writes, reads are occurring far more often than writes, because every process must poll the *entering* and *number* fields of all other processes, but writes just its own. In order to reduce the total amount of memory traffic, this mutex related traffic is a good candidate for revision, as the implementation of synchronization is transparent to the application and can be changed without touching the application, in contrast to cache utilization and shared data accesses.

Figure 4 shows additional information about the state of the mutexes of the same applications. The figure shows that mutexes of `fluidanimate` are (almost) always free at the moment they are being locked, where mutexes of the other application are free for about 80% of the time. Additionally, the hatched area shows the fraction of mutexes that where not only free, but also classified as *relocks*: a successive lock on the same mutex by the same process.

Based on Figure 4, it can be concluded that most of the mutexes are usually free and reused by the same process, so a mutex is (mostly) *local*. Busy mutexes, for which processes are blocking each other, are scarce. Since they involve (expensive) global synchronization, they should be avoided or removed, but this is out of the scope of this paper.

As shown above, mutexes contribute to the memory bandwidth usage, which is a scarce resource in many-core SoCs. The proposed solution implements locks on another infrastructure, which bypasses the SDRAM completely and exploits the locality of mutexes. Since mutexes only require a small amount of memory, they can be kept locally, in a (non-coherent) cache, for example.

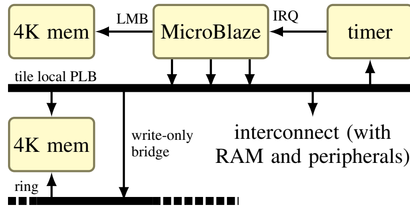


Fig. 5: MicroBlaze tile with additional ring

V. Ring and Distributed Lock

In order to implement inter-processor communication to bypass main memory, the architecture as described in Section III is extended with an additional interconnect: a ring topology, which is depicted in Figure 5. The ring allows write-only access to the local memory of all other MicroBlazes, where a MicroBlaze can read and write its own local memory. In hardware, the ring transfers the destination address and data. It is built of one register, a MUX and comparator per tile, such that every clock cycle one word is transferred to its neighbor register when the address is not within the range of the current tile. In addition to the synthesis results of Section III-C, the implementation of the ring requires one BRAM per tile and adds about 1.4% logic for the PLB slaves and the ring itself. The network can be kept low-cost, because 1) the required bandwidth for the synchronization is low; 2) the routing in a ring is trivial, and thus cheap; and 3) the latency of one clock cycle per tile is not an issue, because most of the latency is introduced by the software of the message handling daemons—even when 100 cores are added, the increased latency by the ring is much less than a single context switch to the daemon.

All accesses over the ring are memory-mapped—the system is a distributed shared memory (DSM) architecture. On top of the ring and local memories, a message-passing API has been implemented, allowing all-to-all communication between the cores. Inter-process messages are sent and received via a message handling process (one per MicroBlaze) that schedules outgoing messages and dispatches incoming messages to the appropriate local process. All messages are processed sequentially by this daemon in FCFS manner (or round-robin in case of simultaneous arrival). The message-passing implementation is a generic interface and is not specifically tailored to synchronization.

Based on message-passing, a new asymmetric distributed lock has been designed, where 'asymmetric' points to the different roles of participants in the algorithm. The main idea is that when a process wants to enter a critical section—and tries to lock a mutex—it sends a request to a server. This server either responds with "You got the lock and you own it", or "Process n owns the lock, ask there again". In the latter case, the locking process sends a message to n, which can reply with "The lock is free, now you own it", or "It has been locked, I'll signal you when it is unlocked". When a process unlocks a lock, it will migrate it to the process that

Algorithm 1 Lock server

Global: M : lock $\rightarrow$ process, $M \leftarrow \varnothing$

Procedure: request(ℓ, r)
Input: process r requesting lock ℓ
1: **if** $M(\ell)$ is undefined **then**
2: $M(\ell) \leftarrow r$
3: **return** got lock ℓ
4: **else**
5: $p \leftarrow M(\ell)$
6: $M(\ell) \leftarrow r$
7: **return** ask p for state of ℓ

Procedure: giveup(ℓ, r)
Input: process r giving up lock ℓ
8: **if** $M(\ell) = r$ **then**
9: undefine $M(\ell)$

Algorithm 2 Message handling daemon

Global: administration of owned locks of local process p:
L_p : lock $\rightarrow$ {free,locked,migrate,stolen}, $\forall p : L_p \leftarrow \varnothing$

Procedure: ask(ℓ, p, r)
Input: lock ℓ, owned by process p, requested by r
1: **atomic**
2: **if** $L_p(\ell)$ is undefined **then**
3: **return** free
4: **else if** $L_p(\ell) =$ locked **then**
5: $L_p(\ell) \leftarrow$ migrate on unlock to r
6: **return** locked
7: **else**
8: $L_p(\ell) \leftarrow$ stolen
9: **return** free

Algorithm 3 Locking process

Global: L_{self}, which is one of L_p of Algorithm 2

Procedure: lock(ℓ)
Input: lock ℓ
1: **atomic**
2: $s \leftarrow L_{\text{self}}(\ell)$
3: $L_{\text{self}}(\ell) \leftarrow$ locked
4: **if** $s \neq$ free **then**
5: **if** request(ℓ,self) $=$ ask p **then**
6: **if** ask(ℓ,p,self) $=$ locked **then**
7: wait for signal (signal counterpart at line 14)

Procedure: unlock(ℓ)
Input: locked lock ℓ
8: dummy read SDRAM
9: **atomic**
10: $s \leftarrow L_{\text{self}}(\ell)$
11: $L_{\text{self}}(\ell) \leftarrow$ free
12: **if** $s =$ migrate to r **then**
13: undefine $L_{\text{self}}(\ell)$
14: signal r (wait counterpart at line 7)
15: **else if** too many free locks in L_{self} **then**
16: $\ell' \leftarrow$ oldest free lock in L_{self}
17: undefine $L_{\text{self}}(\ell')$
18: giveup(ℓ',self)

asked for it, or flag it is being free in the local administration otherwise. In this way, processes build a fair, distributed, FCFS waiting queue. In great contrast to a token-based solution, which also migrates ownership of locks, processes only give up the ownership when they are asked for it. Algorithms 1 to 3 show the implementation. In more detail:

- *lock server* (Algorithm 1): a process that registers the owner process of a lock (or the last one waiting) using the map M. When the server gets a *request* message, it responds with either that the lock is available (line 3) or already owned (line 7). When a process owning a lock does not need it anymore, it can *give it up*. A lock is (statically) assigned to a single server and a server can service many locks.
- *message handler* (Algorithm 2): every MicroBlaze runs a single message handling daemon, which handles incoming messages (see above). When another process *asks* for an owned lock, this daemon inspects the lock administration (denoted map L_p) of the owning local process. Then, it either marks the lock for migration on unlock (line 5) or steals the lock (line 8). In case of the race condition that the give up and the ask message are sent concurrently, the daemon replies that the lock is free (line 3).
- *locking process* (Algorithm 3): the process that wants to enter a critical section. When *locking*, it checks its own administration. When the lock is already owned by the process, it will lock it immediately, without communicating with other cores. Otherwise, it will request the server (line 5) and ask the owner (line 6) of the lock when appropriate. Asking for a locked lock implicitly enqueues the asking process (line 7). On *unlock*, an (uncached) read from main memory is performed (line 8), which enforces an ordering between communication via the ring and operations on the background memory and ensures that all outstanding memory operations will be completed before the lock is unlocked—the system implements the release consistency memory model. This is guaranteed, as the interconnect arbitrates in FCFS manner and the memory controller processes all requests in-order. Then, only locally is the state updated (line 11), unless there is a process already waiting, which will be signaled in that case (line 14). When the process exits, all owned locks must be given up, which is left out of Algorithm 3 for simplicity.

The performance of the proposed ring and new lock algorithm will be compared to the bakery lock below.

VI. Bakery vs. Distributed Lock

To evaluate the distributed lock, experiments are conducted on two systems: the bakery implementation in the architecture without ring (Section III), referred to as the base case; and the distributed lock via the ring (Section V).

For the latter, the maps M and L_p of Algorithms 1 to 3 are implemented using AA trees [18], having $O(\log n)$ complexity, where n is the number of owned locks. Although the different concepts of a lock server and message handling daemon are important in the algorithm, the functionality of the server is merged into the daemon in the actual implementation. This allows a quick response of request and give up messages. As a result, every message handling daemon can act like a lock server and locks are statically assigned to one of the 32 daemons based on the address of the mutex, which implements a naive way of load balancing.

The four applications have been run on both systems, repeated ten times with slightly different compile settings to average out cache effects by placing memory segments differently. All applications start one worker process on each of the 32 cores. When a process blocks on a mutex, the blocked time is left unused by the application for that specific core. Only the parallel body of the application has been measured; (sequential) preprocessing steps have been ignored.

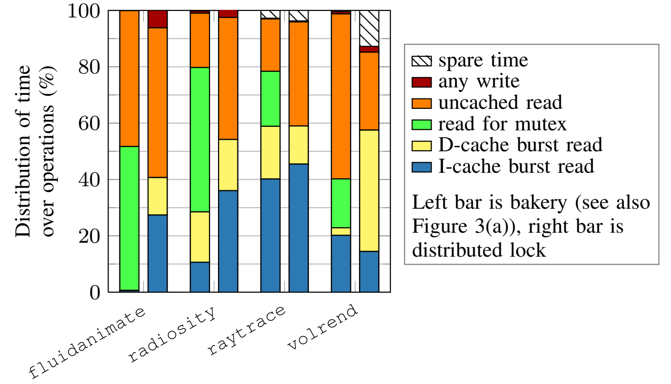


Fig. 6: Measured traffic on SDRAM controller, using bakery and distributed locks

TABLE II: Distributed lock statistics

	# locks	request[a]	ask[a]	giveup[a]	signals[a]	msgs/s[b]
fluidanimate	2,935,247	285,176	280,665	4,511	1,110	25,898.3
radiosity	1,594,306	627,154	574,620	52,533	214,764	8,480.6
raytrace	33,831	1,681	1,645	36	73	102.0
volrend	56,380	33,962	33,886	76	8,024	3,892.9

[a] See Algorithms 1 to 3
[b] Lock messages exchanged per second over the ring

A. Results

Figure 6 shows the types of memory traffic of both the base case and the distributed lock. The figure shows that for `raytrace` and `volrend` the memory bandwidth is not saturated. In Table II, the measured number of exchanged messages is depicted. Even for `fluidanimate`, which is quite message-intensive, the ring utilization is very low; one message consists of eight words, where the ring allows injection of one word per core every clock cycle.

Performance numbers, which are averaged over all runs, of the applications are shown in Figure 7. All values are normalized to the base case, which is 1 by definition. In the chart, the first metric shows the relative change of the execution time of the application: all application benefit from

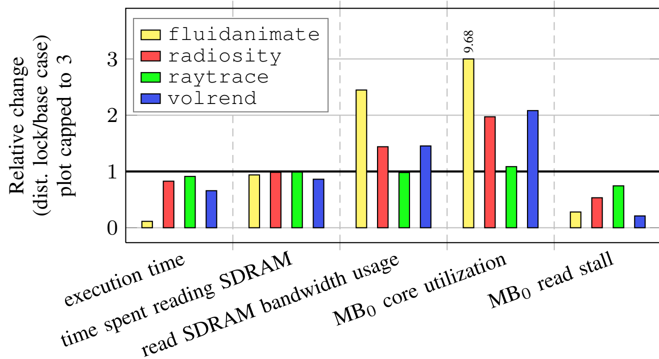


Fig. 7: Difference between using bakery algorithm and asymmetric distributed lock with ring

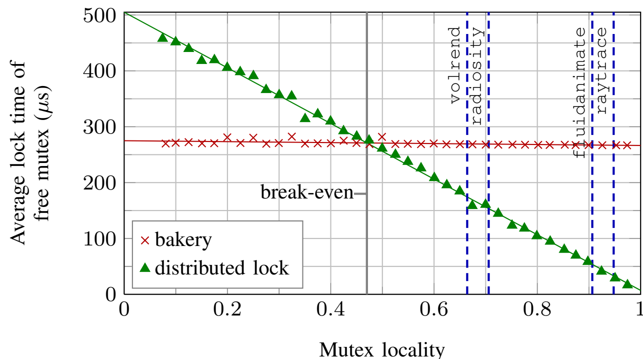


Fig. 8: Impact of locality on acquiring time of a mutex lock, based on synthetic benchmark

the distributed lock, `fluidanimate` is even 9 times faster than using the bakery lock.

Next, two metrics indicate the SDRAM usage, which aggregates operations of all 32 cores. It shows that the memory bandwidth is used more effectively; the read bandwidth increases, with a similar time spent on reading, because less uncached word and more burst operations are performed.

Two metrics are shown of only one MicroBlaze that has hardware support for measuring micro-architectural events. Since all cores are running the same kind of workload, it can be assumed that all other cores behave similarly. Overall, the core utilization is higher for all applications, since cores stall less on (uncached) reads.

B. Discussion

Whether the complexity of the distribution lock pays off for a given application, is closely related to the locality of its mutexes. There is a trade-off between a main memory polling algorithm like bakery that consumes scarce bandwidth, or keeping (or caching) mutexes locally and having higher latencies for non-local mutexes. Figure 8 gives insight into this trade-off.

Naturally, when a mutex is used by only one process, it is always local and locking it is very fast. When mutexes are used by more processes, the lock must be migrated regularly, which involves communication between cores. Hence, the amount of expensive communication depends on the *locality* of the mutex, which we define as the fraction of relocks over free locks. In the synthetic setup used for Figure 8, a mutex is forced to a specific locality, and the average time is measured of locking that mutex. The figure shows the relation between locality and average lock time for both the bakery implementation (which takes 270 μs on average) and the distributed lock (3.5 μs for a relock). Although the exact slope and height of the lines in the figure depend on the workload, the trend is always the same.

For the four applications, the locality (as can also be found in Figure 4) is respectively 0.91, 0.71, 0.95, 0.66, and is also indicated in Figure 8. This shows that applications with globally used mutexes, possibly do not benefit from the distributed lock[1], but the tested applications are all at the right side of the break-even point.

Calculations indicate that the applications still spent respectively 35%, 26%, 0.2% and 9.2% of the execution time on mutex locking operations. Additionally, during this locking time, 77%, 95%, 85% and 95% of the time is spent waiting for locked mutexes. Hence, improving the locking speed any further will hardly lead to a performance increase; making locks more local is probably more beneficial.

Although the distributed lock has only been tested on 32 cores, we expect that the costs and trade-off are the same on larger systems. As Algorithms 1 to 3 do not depend on the number of cores, just the number of stored locks in maps M and L_p influence the performance. When the balance between servers and worker processes is kept the same, then the concurrency in the design of the application is the only relevant factor.

VII. Conclusion

This paper presents an asymmetric distributed lock algorithm for non-cache coherent embedded multiprocessor systems. Experiments have been carried out with a 32-core MicroBlaze system realised on a Virtex-6 FPGA on a ML605 evaluation board, which has an SDRAM memory bank.

In our solution, mutex related SDRAM memory traffic is completely eliminated and replaced by synchronization messages via a low-cost write-only ring interconnect. This ring only increases the hardware resources by 1.4%. Measurements indicate that of all tested applications, the locality of mutexes is high. Our asymmetric lock benefits from this locality and reduces the time for relocking from 270 μs to 3.5 μs compared to an implementation of the bakery algorithm. For our benchmark set, the maximum reduction in execution time was 89% and the average reduction was 37% compared to the use of the same multiprocessor system without the use of asymmetric lock and ring. Additionally, these experiments show that although any-to-any communication can be realized via main memory, the addition of a low-cost ring is beneficial, even when it is only used for synchronization.

[1]Obviously, one can argue that having global locks in massively parallel applications is a bad programming habit anyway.

VIII. References

[1] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, February 1991.

[2] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, 1998, ISBN 1558603433.

[3] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 124–149, January 1991.

[4] J. Howard, S. Dighe, Y. Hoskote *et al.*, "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," in *ISSCC 2010*.

[5] F. Ophelders, M. J. G. Bekooij, and H. Corporaal, "A tuneable software cache coherence protocol for heterogeneous MPSoCs," in *Proceedings of CODES+ISSS '09*. ACM, pp. 383–392.

[6] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, pp. 453–455, August 1974.

[7] M. Singhal, "A taxonomy of distributed mutual exclusion," *Journal of Parallel and Distributed Computing*, vol. 18, no. 1, pp. 94–101, 1993.

[8] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, "Hardware Synchronization for Embedded Multi-Core Processors," in *Proc. of ISCAS 2011*.

[9] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Power/Performance Hardware Optimization for Synchronization Intensive Applications in MPSoCs," in *Proceedings DATE '06*.

[10] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *Proceedings of ISCA '07*. ACM, pp. 35–45.

[11] A. Tumeo, C. Pilato, G. Palermo, F. Ferrandi, and D. Sciuto, "HW/SW methodologies for synchronization in FPGA multiprocessors," in *Proceeding of the ACM/SIGDA int'l symposium FPGA '09*, pp. 265–268.

[12] J. Abellán, J. Fernández, and M. Acacio, "A G-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs," in *2010 39th Int'l Conf on Parallel Processing*. IEEE, 2010, pp. 267–276.

[13] C. Yu and P. Petrov, "Distributed and low-power synchronization architecture for embedded multiprocessors," in *Proceedings of CODES+ISSS '08*. ACM, 2008, pp. 73–78.

[14] M. Wu and W. Shu, "An Efficient Distributed Token-Based Mutual Exculsion Algorithm with Central Coordinator," *Journal of Parallel and Distributed Computing*, vol. 62, no. 10, pp. 1602 – 1613, 2002.

[15] M. R. Casu, M. R. Roch, S. V. Tota, and M. Zamboni, "A NoC-based hybrid message-passing/shared-memory approach to CMP design," *Microprocessors and Microsystems*, vol. 35, no. 2, pp. 261 – 273, 2011.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proc. Symp. nd Annual Int Computer Architecture*, 1995.

[17] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[18] A. Andersson, "Balanced search trees made simple," *Algorithms and Data Structures*, pp. 60–71, 1993.